A Certified Core Policy Language

Bahman Sistany and Amy Felty

School of Electrical Engineering and Computer Science, University of Ottawa, Canada

Abstract. We present the design and implementation of a Certified Core Policy Language (ACCPL) that can be used to express access-control policies. We define formal semantics for ACCPL where we discover and enumerate all possible cases when answering an access query. We use the Coq Proof Assistant to state theorems about the semantics of ACCPL, to develop proofs for those theorems and to machine-check the proofs ensuring correctness guarantees are provided. The main design goal for ACCPL is the ability to reason about the policies written in ACCPL with respect to specific questions. In addition ACCPL is designed to be extendable so that extensions to expressive power may be explored with respect to the established reasonability properties. To this end, ACCPL is small (the syntax and the semantics of ACCPL only take a few pages to describe), although we believe ACCPL supports the core features of access-control policy languages.

Keywords: program correctness, formal verification, access control, policy analysis, Coq, XACML, DRM, ODRL, SELinux

1 Introduction

We describe the design of a Certified Core Policy Language (ACCPL) and its implementation in the Coq Proof Assistant. Using Coq to implement ACCPL was an important factor in its design, allowing us to address the trade-off between expressive power and ease of formal proof of correctness. The semantics of ACCPL are specified by translation from policy statements together with an access request and an environment containing all the relevant facts, to decisions. We present results showing the translation functions behave correctly with respect to the decision question that asks whether a request to access a resource may be granted or denied, given a policy. The translation functions also cover the case where a given policy does not apply to a request in which case a decision of non-applicable is rendered. Our results show that for each access request, the translation algorithm terminates on all input policies with a decision of granted, denied or non-applicable. Proving that the translation algorithm terminates on all input policies (given an access request) with a decision of granted, denied or non-applicable, is the specific goal based on which the semantics were defined.

To motivate the design of ACCPL an access-control policy language, let us review the definition of “access-control”: Authorization refers to the process of rendering a decision to allow access to a resource or asset of interest. By the same token all unauthorized access requests to resources must be controlled and ultimately denied, hence the term “access-control”.

Although various access-control models exist, to harmonize access control in large environments with many subjects and objects and disparate attributes, Policy-based Access Control (PBAC) model has been proposed. PBAC allows for a more uniform access-control model across the system. There is also a need for large organizations to put in place mechanisms such that access-control rules can be easily audited. This calls for a data-driven approach to access-control where the data, in this case the access-control rules, are available to read and analyze.

Because of the cited advantages of PBAC and its generality and wide spread use, PBAC is the model ACCPL implements.

1.1 A Core Policy Language for PBAC Systems

Currently the most popular Rights Expression Languages (REL)s are eXtensible rights Markup Language (XrML) [19], and Open Digital Rights Language (ODRL) [8]. Both of these languages are XML based and are considered declarative languages. RELs, or more precisely Digital Rights Expression Languages (DREL)s when dealing with digital assets deal with the “rights definition” aspect of the Digital Rights Management (DRM) ecosystem. A DREL, allows the expression and definition of digital asset usage rights such that other areas of the DRM ecosystem, namely the enforcement mechanism and the usage tracking components can function correctly.

DRM refers to the digital management of rights associated with the access or usage of digital assets. There are various aspects of rights management however. According to the authors of the white paper “A digital rights management ecosystem model for the education community” [5] digital rights management systems comprise these categories: defining rights, distributing/acquiring rights, enforcing rights and finally tracking usage.

eXtensible Access Control Markup Language (XACML) is a high-level and platform independent access control system that is also XML based. No formal semantics is provided for XACML similar to both XrML and ODRL. The XACML standard is written in prose and contains quite a number of loose points that may give rise to different interpretations and lead to different implementation choices [11].

XACML, ODRL and XrML are all PBAC based languages where ODRL and XrML differ from XACML by their focus on digital assets protection and in general DRM, hence the term REL. All three are full-blown and custom languages that have one thing in common; they suffer from a lack of formal semantics. Additionally all of these languages cover much more than policy expressions leading to access decisions; they also address enforcement of policies (ODRL and XrML specifically and DRM in general distinguish themselves from general access-control languages by additionally addressing enforcement of policies beyond where the policies were generated). A third reason that made these custom languages unsuitable as a core policy language was the fact that they are limited in terms of what can be built on top of them; for example expressing hierarchical role-based access-control in XACML requires a fairly complex encoding [18].

Rights expressions in DRELs and specifically ODRL are used to arbitrate access to assets under conditions which is very similar to how access control conditions are expressed in access control policy languages such as XACML and Security Enhanced Linux (SELinux) [15]. In fact several authors have worked on interoperability between RELs and access control policy languages, specifically between ODRL and XACML [13,10].

A policy language based on logic and formal semantics but one that was small and extendible was needed. We use Pucella and Weissman's subset of ODRL [14] as the starting point for ACCPL and in doing so treat digital rights as our main access-control application without loss of generality with respect to other applications, with the final goal of performing formal verification on policies written in ACCPL.

1.2 Formal Semantics for PBAC Languages

Formal methods help ensure that a system behaves correctly with respect to a specification of its desired behaviour [12]. This specification of the desired behaviour is what's referred to as *semantics* of the system. Using formal methods requires defining precise and formal semantics, without which analysis and reasoning about properties of the system in question would become impossible. To formalize the semantics of PBAC languages several approaches have been attempted by various authors. Most are logic based [6,14] while others are based on finite-automata [7], operational semantics based interpreters [**?**] and web ontology (from the Knowledge Representation Field) [9].

1.3 Specific Problem

Policy languages and the agreements written in those languages are meant to implement specific goals such as limiting access to specific assets. The tension in designing a policy language is usually between how to make the language expressive enough, such that the design goals for the policy language may be expressed, and how to make the policies verifiable with respect to the stated goals.

As stated earlier, an important part of fulfilling the verifiability goal is to have formal semantics defined for policy languages. For ODRL, authors of [14] define a formal semantics based on which they declare and prove a number of important theorems (their main focus is on stating and proving algorithm complexity results). However as with many paper-proofs, the language used to do the proofs while mathematical in nature, uses many intuitive justifications to show the proofs. As such these proofs are difficult to verify or to "derive". Furthermore the proofs can not be used directly to render a decision on a sample policy (e.g. whether to allow or deny access to an asset). Of course one may (carefully) construct a program based on these proofs for practical purposes but certifying such programs correct presents additional verification challenges, even assuming the original proofs were in fact correct.

1.4 Contributions

We designed a policy based access-control language called ACCPL based on ODRL and starting with definitions in [14]. The ACCPL framework has been encoded in *Coq* [1] which is both a programming language and a proof-assistant. We have specified and proved ACCPL correct with respect to properties of interest (see section 4.3) in Coq which will allow us to extract programs from the proofs; the executable programs can be used on specific policies and a query, to render a specific decision such as "a permission has been granted".

We originally started with a specific subset of [14] so that we could concentrate on what we believed to be the essence or core of the language. Initially we intended to maintain the central semantic definitions including "Closed World Assumptions" [14] where the semantics only specify explicitly Permitted and NotPermitted answers however we discovered the semantics as stated by Pucella and Weissman [14] are not explicit and therefore the decision question that asks whether a request to access a resource may be granted or denied, may not be answered in all cases. We have therefore made major modifications to the semantics of Pucella and Weissman's language such that an answer to a request for access to a resource may be determined unambiguously and for all cases.

Our results subsume an important sub-category, namely inconsistency or conflict-detection in policy expressions or rules. St-Martin and Felty [16] describe and implement in Coq a conflict detection algorithm for detecting conflicts in XACML access control rules. XACML is an expressive and at the same time complex policy language which makes conflict detection a difficult task. The authors of [16] then prove the conflict detection algorithm correct (or certified) by developing a formal proof in Coq. The proof is rather complex and involves a large number of cases, including many corner cases that were difficult to get right [16]. For ACCPL we have formally proven that conflicts are not possible.

Given that ACCPL is a core policy language with semantics that have been certified correct, we could use ACCPL to implement various (more expressive) policy languages. In addition ACCPL could be used as an intermediate language to reason about interoperability between those policy languages [13,10]. In this manner our language ACCPL can be viewed as a extendable language, complete with defined and verified semantics, that can be used as the basis for implementing various policy languages with more expressive power (e.g. W3C's ODRL and SELinux).

For access to the Coq source code for ACCPL, please refer to http://www.site.uottawa.ca/~afelty/accpl/.

2 ACCPL

We follow the style of [14] by using abstract syntax to express policy statements in ACCPL.

2.1 Environmental Facts

To determine the outcome of policies, specified conditions in those policies are evaluated but to do so environmental facts are often needed. In the DRM realm with its focus on usage control, certain facts are typically tracked in the environment. The count of how many times an asset has been accessed, the amount a user has paid to access an asset and finally whether a user has made an attribution (e.g. mentioning the content owner by name) are examples of the kind of facts environments hold. In ACCPL, agreements and facts (i.e. environments) will refer to a count of how many times each policy should be used and has been used respectively, to justify an action.

2.2 Abstract Syntax for ACCPL

Listing 1.1. Abstract Syntax for ACCPL

```
<agreement> ::=
     'agreement' 'for' <prin> 'about' <asset> 'with' <
        policySet>
<prin> ::=  { <subject_1>, ..., <subject_m> }
<asset> ::= TheReport | ebook | latestJingle | ...
<subject> ::= Alice | Bob | ...
<act> ::= Play | Print | Display | ...
<policySet> ::=
  <primPolicySet>                   ; primitive policy set
<primPolicySet> ::=
   <primInclusivePolicySet>         ; primitive inclusive policy set
  |<primExclusivePolicySet>         ; primitive exclusive policy set
<primExclusivePolicySet> ::=
  <prerequisite> ↦ <policy>         ; primitive exclusive policy set
<primInclusivePolicySet> ::=
  <prerequisite> → <policy>         ; primitive inclusive policy set
<primPolicy> ::=
  <prerequisite> ⇒_<policyId> <act>           ; primitive policy
<policy> ::=
  'and'[ <primPolicy_1>, ...,
      <primPolicy_m> ]                        ; conjunction
 <policyId> ::= N
<primPrerequisite> ::=
    'True'                          ; always true
  |  <constraint>                   ; constraint
  |  'not' [ <constraint> ]         ; suspending constraint
<prerequisite> ::=
  |  'and'[ <primPrerequisite_1>, ...,
    <primPrerequisite_m> ]          ; conjunction
<constraint> ::=
```

```
    <prin>                              ; principal
  |  'Count' [N]                        ; number of executions
  |  <prin> ('Count' [N])               ; number of executions by prin
```

The abstract syntax for ACCPL is given in Listing 1.1. The top level production is the `<agreement>`. An agreement expresses what actions a set of subjects may perform on an object and under what conditions. Syntactically an agreement is composed of a set of subjects called a principal or `<prin>`, an `<asset>` and a `<policySet>`. Principals (`<prin>`) are composed of `<subject>`s which are specified based on the application e.g. `Alice`, `Bob`, etc. Assets and actions are also application specific such as `TheReport` and `ebook` for assets and `Display` and `Print` for actions.

A policy set is a primitive policy set implying non-nested policy sets. Each primitive policy set specifies a `<prerequisite>` and a `<policy>`. Intuitively if the prerequisite "holds" the policy is taken into consideration. Otherwise the policy will not be looked at. Some primitive policy sets are specified as inclusive as opposed to others that are explicitly specified as exclusive. Primitive exclusive policy sets are exclusive to an agreement's users in that only those users may perform the actions specified in the policy set. The implication is that all other users who are not specified in the agreement's principal are forbidden from performing the specified actions, no matter whether the prerequisite holds or not. Not surprisingly we also define primitive inclusive policy sets that don't enforce any exclusivity to the agreement's users.

A primitive policy specifies an action to be performed on an asset, depending on whether the policy's prerequisite holds or not. If the prerequisite holds the agreement's user is permitted to perform the action on the agreement's asset; otherwise permission is denied. A unique identifier for each policy to help the translation (from agreements to formulas), called the policy identifier (`<policyId>`) is included in our definition of the policy construct, however, as far as the proofs are concerned the policy identifier could be removed without a loss to the obtained results. A policy is made up of primitive policies. Primitive policies are grouped together using the conjunction combining operator.

In ACCPL a `<primPrerequisite>` is either `True` or it is a `<constraint>`. The True prerequisite always holds. A constraint is an intrinsic part of a policy and cannot be influenced by agreement's users. A constraint can also be negative, specified by the keyword 'not' in front of `<constraint>`. A `<prerequisite>` is a set of primitive prerequisites which are closed under conjunction operator specified by the keyword 'and' in front of the list of the primitive prerequisites separated by commas.

Constraints are either of the principal kind, the count kind, or the count by principle kind. Principal constraints require matching to the users listed following the keyword `<prin>`. For example, the constraint of "the user being Alice" is a constraint of type principal. A count constraint refers to the number of times the user of an agreement has invoked policies to justify her actions. If the count constraint is part of a policy then the count refers to that single policy. In the case that the count constraint is part of a policy set or if the policy is a conjunction, then the count refers to the set of policies specified in the policy set or in the policy conjunction as the case may be.

2.3 ACCPL Syntax in Coq

ACCPL productions were presented as high level abstract syntax in Section 2.2. We present the corresponding encodings in Coq in Listing 1.2. Note that the data type `nonemptylist` represents a list data structure that has at least one element and data types `asset`, `subject`, `act` and `policyId` are simply defined as Coq's `nat`.

Listing 1.2. ACCPL: Coq Version of Agreement

```
Inductive agreement : Set :=
  | Agreement : prin → asset → policySet → agreement.
Definition prin := nonemptylist subject.
Inductive policySet : Set :=
  | PPS : primPolicySet → policySet.
Inductive primPolicySet : Set :=
  | PIPS : primInclusivePolicySet → primPolicySet
  | PEPS : primExclusivePolicySet → primPolicySet.
Inductive primInclusivePolicySet : Set :=
  | PrimitiveInclusivePolicySet : preRequisite → policy → primInclusivePolicySet.
Inductive primExclusivePolicySet : Set :=
  | PrimitiveExclusivePolicySet : preRequisite → policy  → primExclusivePolicySet.

Inductive policy : Set :=
  | Policy : nonemptylist primPolicy → policy.
Inductive primPolicy : Set :=
  | PrimitivePolicy : preRequisite → policyId → act → primPolicy.
Inductive primPreRequisite : Set :=
  | TruePrq : primPreRequisite
  | Constraint : constraint → primPreRequisite
  | NotCons : constraint → primPreRequisite.
Inductive preRequisite : Set :=
  | PreRequisite : nonemptylist primPreRequisite → preRequisite.
Inductive constraint : Set :=
  | Principal : prin  → constraint
  | Count : nat → constraint
  | CountByPrin : prin → nat → constraint.
```

We now show the statement expressing "the asset TheReport may be printed a total of 2 times by Alice only" in the abstract syntax notation in 1.3, as encodings of ACCPL constructs in Coq in 1.4 and finally as an ACCPL construct in 1.5.

Listing 1.3. First Agreement for Alice and Bob

```
agreement
 for Alice and Bob
 about The Report
 with True → and[Alice, count[2]] ⇒_id1 print.
```

Listing 1.4. Expressing First Agreement for Alice and Bob in ACCPL

```
Definition ps_xml_p1prq1:primPreRequisite :=
  (Constraint (Principal (Single Alice))).
Definition ps_xml_p1prq2:primPreRequisite :=
  (Constraint (Count 2)).
Definition ps_xml_prq:preRequisite :=
  (PreRequisite (NewList ps_xml_p1prq1 (Single ps_xml_p1prq2))).
Definition ps_xml_p1:primPolicy :=
  (PrimitivePolicy ps_xml_prq id1 Print).
Definition ps_xml_p:policy :=
  (Policy (Single ps_xml_p1)).
Definition ps_xml:primPolicySet :=
  PIPS (PrimitiveInclusivePolicySet
    (makePreRequisite TruePrq) ps_xml_p).
Definition Axml := Agreement (NewList Alice (Single Bob)) TheReport (PPS
      ps_xml).
```

Listing 1.5. Fully Built First Agreement for Alice and Bob in ACCPL

```
Agreement (Alice, [Bob]) TheReport
       (PPS
         (PIPS
           (PrimitiveInclusivePolicySet (PreRequisite [TruePrq])
             (Policy
               [PrimitivePolicy
                 (PreRequisite
                   (Constraint (Principal [Alice]),
                    [Constraint (Count 2)])) id1 Print]))))
```

2.4 Semantics

We specify the semantics of ACCPL as a translation function from an agreement together with an access request and an environment containing all relevant facts, to decisions. This is done based on whether there are proof terms for certain conditions and/or proof terms for the negation of those conditions. The translation functions plus the auxiliary types and infrastructure which implement the semantics for ACCPL have been encoded in Coq.

The `sumbool` type is a boolean type defined in the Coq standard library; it captures the idea of program values that indicate which of two propositions is true [4]. The `sumbool` type is equipped with the justification of their value [17] which help with proofs. We have used the `sumbool` type to declare and prove decision procedures that we have subsequently used in the translation functions implementing the semantics and also in the proofs.

2.5 Types of Decisions and their Implementation in Coq

Policy based access-control languages typically use a two-valued decision set to indicate whether an access request is granted or denied. When a decision for a query is not granted, one design choice for a language is to return an explicit deny decision. However in this case deny stands for "not permitted". It is possible to have cases when the policy truly doesn't specify either a permit or a deny decision. In such cases arbitrarily returning the decision of deny makes it difficult to compose policies and in fact, an explicit decision of "non applicable" is warranted in such cases. Some languages may decide to only support permit decisions. In such languages lack of a permit decision for a query signifies a deny decision so deny decisions are not explicit. Although the policies of these languages may be more readable than those with more explicit decisions, they result in ambiguity on whether a deny decision was really intended or not. Finally some languages define an explicit decision of "error" for cases such as when both permit and deny decisions are reached for the same query. An explicit error decision is preferable to undefined behaviour because it can lead to improvements to policies and/or how the queries are built [18]. In ACCPL we use a three-valued decision set: `Permitted`, `NotPermitted` and `Unregulated` (used as synonymous with "non-applicable").

2.6 Translations

Intuitively a query or request asks the following question given an agreement: "May subject `s` perform an action `ac` to asset `a`?". We represent a query by its components, namely the subject, action and asset that form the query question: `action_from_query`, `subject_from_query` and `asset_from_query`.

In the following we present the high-level description of how the main algorithm (encoded in the translation functions) works in two separate listings based on whether the policy set in question is inclusive or exclusive.

The first listing (1.6) for inclusive policy sets, shows how a positive answer to a query in the form of a `Permitted` decision is reached. All cases when a decision of `Unregulated` is rendered are explicitly captured and shown. The second listing (1.7) for exclusive policy sets, shows how a negative answer to a query in the form of a `NotPermitted` decision is reached. This listing also shows that a positive decision of `Permitted` is reached in exactly the same way as the case for inclusive policy sets. All cases when a decision of `Unregulated` is rendered are explicitly captured and shown.

Listing 1.6. Access Decision Pseudocode: Inclusive Policy Sets

```
IF (asset_from_query = asset_from_agreement)
  IF (subject_from_query is IN prin_u)
    IF (The preRequisite from the policy set HOLDS)
      IF (The preRequisite from the policy HOLDS)
        IF (action_from_query = action_from_agreement)
          result = subject_from_query is Permitted to perform
 action_from_query on asset_from_query
        ELSE
          result = Unregulated
        END_IF
      ELSE
```

```
        result = Unregulated
      END_IF
    ELSE
      result = Unregulated
    END_IF
  ELSE
      result = Unregulated
  END_IF
ELSE
  result = Unregulated
END_IF
```

Listing 1.7. Access Decision Pseudocode: Exclusive Policy Sets

```
IF (asset_from_query = asset_from_agreement)
  IF (subject_from_query is IN prin_u)
    IF (The preRequisite from the policy set HOLDS)
      IF (The preRequisite from the policy HOLDS)
        IF (action_from_query = action_from_agreement)
          result = subject_from_query is Permitted to perform
  action_from_query on asset_from_query
        ELSE
          result = Unregulated
        END_IF
      ELSE
        result = Unregulated
      END_IF
    ELSE
      result = Unregulated
    END_IF
  ELSE
        IF (action_from_query = action_from_agreement)
          result = subject_from_query is NotPermitted to perform
  action_from_query on asset_from_query
        ELSE
          result = Unregulated
        END_IF
  END_IF
ELSE
  result = Unregulated
END_IF
```

2.7 Correctness of ACCPL

The theorem `trans_agreement_dec2` is the declaration of the main correctness result for ACCPL (see listing 1.9). Together with proofs for other theorems and lemmas, we have "certified" ACCPL correct by proving this theorem. The nonempty list that the agreement translation function `trans_agreement` returns will contain results one per each primitive policy (primPolicy) found in the agreement. Specifically the predicate isResultInQueryResult takes a result and a nonempty list of result's which trans_agreement produces, and calls the In predicate. The In predicate checks for the existence of the input result in the nonempty list of results.

Note that by mentioning the agreement translation function directly in the statement of the theorem 1.9, we tie the correctness property to how the translation functions work. To prove the theorem and with each successive subgoal during the interactive proof process, the definition of the translation function in scope gets unfolded and used so the translation functions have to be defined such that each subgoal is discharged and the proof is completed.

As an example and also a visual aid to understanding how queries are answered, see listing 1.8. The isResultInQueryResult predicate looks for a result with an answer of Permitted in the list that trans_agreement has produced, for an agreement for three primitive policies (since the set contains three results). In words, we are asking whether Alice is allowed to print the asset ebook, given a policy.

Listing 1.8. Access Request Resulting in Decision of Permitted

```
isResultInQueryResult
(Result Permitted Alice Print ebook)
[ (Result Unregulated Alice Print ebook) ; (Result Unregulated Alice Print ebook) ;
     (Result Permitted Alice Print ebook) ]
```

In the case where the whole set is not comprised of Unregulated results, we have two mutually exclusive cases. The first case is when the set has a at least one Permitted result; we answer the access query in this case with a result of Permitted (this would be the case in the listing 1.8). The second case is when the set has at least one NotPermitted; we answer the access query in this case with a result of NotPermitted.

Listing 1.9. Agreement Translation's Correctness Property

```
Theorem trans_agreement_dec2:
  ∀
  (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),

 (isResultInQueryResult
    (Result Permitted subject_from_query action_from_query asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query
      asset_from_query))
\/
 (isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query
      asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query
      asset_from_query))
\/
```

```
(~(isResultInQueryResult
   (Result Permitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query
     asset_from_query)) /\
 ~(isResultInQueryResult
   (Result NotPermitted subject_from_query action_from_query
     asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query
     asset_from_query))).
```

Typically most, if not all of the results will be of type `Unregulated`. In the case where all the results are `Unregulated` we answer the access query with a result of `Unregulated`. We show this case indirectly in the theorem in listing 1.9 by stating the set does not contain a `Permitted` result nor a `NotPermitted` result.

2.8 Mutual Exclusivity of Permitted and NotPermitted

The proof for `trans_agreement_not_Perm_and_NotPerm_at_once` establishes that both `Permitted` and `NotPermitted` results cannot exist in the same set returned by `trans_agreement` (see listing 1.10). This result also establishes the fact that in ACCPL rendering conflicting decisions is not possible given an agreement.

Listing 1.10. Permitted and NotPermitted: Mutually Exclusive

```
Theorem trans_agreement_not_Perm_and_NotPerm_at_once:
  ∀
  (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),

 ~((isResultInQueryResult
    (Result Permitted subject_from_query action_from_query asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query
      asset_from_query))
/\
 (isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query
      asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query
      asset_from_query))).
```

The proof for the next theorem `trans_agreement_not_NotPerm_and_-not_Perm`
`_implies_Unregulated_dec` shows that in the case where neither a `Permitted` nor a `NotPermitted` result exists in the set returned by `trans_agreement`, there does exist at least one `Unregulated` result (see listing 1.11).

Listing 1.11. Not (Permitted and NotPermitted) Implies Unregulated

```
Theorem
    trans_agreement_not_NotPerm_and_not_Perm_implies_Unregulated_dec:
 ∀
  (e:environment)(ag:agreement)(action_from_query:act)
  (subject_from_query:subject)(asset_from_query:asset),

  (~(isResultInQueryResult
    (Result Permitted subject_from_query action_from_query asset_from_query)

      (trans_agreement e ag action_from_query subject_from_query
    asset_from_query)) /\

  ~(isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query
    asset_from_query)
      (trans_agreement e ag action_from_query subject_from_query
    asset_from_query))) →

 (isResultInQueryResult
    (Result Unregulated subject_from_query action_from_query
    asset_from_query)
     (trans_agreement e ag action_from_query subject_from_query
    asset_from_query)).
```

3 Conclusion and Outlook

We presented the design and implementation of ACCPL as a small and certifiably correct policy language. ACCPL is a PBAC system that can be used to express general access-control rules and policies. In addition we have defined formal semantics for ACCPL where we have discovered and added all possible cases when answering a query on whether to allow or deny an action to be performed on an asset. We have subsequently used the Coq Proof Assistant to state theorems about the expected behaviour of ACCPL when evaluating a request with respect to a given policy, to develop proofs for those theorems and to machine-check the proofs ensuring correctness guarantees are provided. We have in particular stated, developed and proved a correctness result for the semantics of ACCPL.

We additionally described why certain design choices were made and how they contributed to the ease of reasoning for ACCPL. Admittedly some expressive power present in other access-control policy languages was omitted from ACCPL in order to achieve the reported correctness proofs. For example, in ACCPL we only support base policy sets (policy sets that are not composed of other policy sets) i.e. no combining of base policy sets using conjunctions or other combining operators are supported.

4 Related Work

In the following sections we will review related work and approaches to define semantics for PBAC based languages such that one can determine without any ambiguity whether a permission or prohibition follows from a set of policy statements.

4.1 Lithium

Halpern and Weissman [6] use First-Order Logic (FOL) to represent and reason about policies; policies describe the conditions under which a request to perform an action, such as reading a file, is granted or denied. They restrict FOL to get tractability for answering the query of whether a request to access a resource may be granted or denied, given a policy, and argue that despite the tractability results their language is still expressive. Halpern and Weissman [6] focus on satisfying three requirements in the design of Lithium: expressive enough, tractable enough and usable by non-experts.

4.2 Automata-based Semantics

Holzer, et al [7] give a semantics for ODRL that models the actions that are allowed according to a contract or an agreement. This model is presented in terms of automata. Each trace through the automaton represents a valid sequence of actions for each participant. The states of the automaton encode the state of the license at each point in time, meaning, which actions are allowed at what point considering the actions that have taken place in the past.

4.3 Conflict Detection Algorithms

Capretta, et al [3] present a conflict detection algorithm for the Cisco firewall specification [2] and formalize a correctness proof for it in the Coq proof assistant. The authors present their algorithm in Coq's functional programming language along with access rules and requests which are also encoded in Coq. The authors also prove in [3] that their algorithm finds all conflicts and only the correct conflicts in a set of rules. The algorithm is therefore verified formally to be both sound and complete.

St-Martin and Felty [16] represent policies for a fragment of XACML 3.0 in the Coq proof assistant and propose an algorithm for detecting all conflicts in XACML policies. They state and prove the correctness of their algorithm in the Coq proof assistant. Their XACML subset includes some complex conditions such as time constraints. The authors compare their work with the conflict detection presented in [3] and conclude that conflict detection in XACML is more complex and results in having to consider many cases including many subtle corner cases.

4.4 Future Work

There are a number of directions that can be taken as future work. In the following we list three distinct directions that could be combined at various stages. Tschantz and Krishnamurthi's [18] present a set of "reasonability properties" to analyze the behaviour of policies in light of additional and/or explicit environmental facts and policy growth and decomposition. We conjecture that ACCPL supports these properties: it is deterministic, total, safe, and it has independent composition property and supports a monotonic policy combinator. However, we have not yet certified (using formal proofs) that ACCPL has these properties, as we claim. We defer proving these properties for ACCPL as future work.

Another direction for future work is to explore different ways ACCPL could be made more expressive. For example, we can add various policy combinators and their semantics to ACCPL using the translation function framework. The translation function framework we have developed for ACCPL is meant to keep the delicate balance between addition of expressiveness while maintaining provability of established results.

A design goal for ACCPL was to make it a target language for deploying policies written in other languages. We could capture, implement and study the semantics of these other policy-based access-control systems using the ACCPL translation function framework and ultimately certify the semantics of those languages with respect to their specifications the same way ACCPL has been certified correct. For example, we can take another PBAC system such as XrML and ODRL, implement them in Coq as additional (or modifications of) existing ACCPL constructs, analyze and reason about them, etc.

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer Science & Business Media (2004), http://www.labri.fr/perso/casteran/CoqArt/index.html
2. Boney, J.: Cisco IOS - in a nutshell: a desktop quick reference for IOS on IP networks (2. ed.). O'Reilly (2005), http://www.oreilly.de/catalog/cisconut2/index.html
3. Capretta, V., Stepien, B., Felty, A.P., Matwin, S.: Formal correctness of conflict detection for firewalls. In: Ning, P., Atluri, V., Gligor, V.D., Mantel, H. (eds.) Proceedings of the 2007 ACM workshop on Formal methods in security engineering, FMSE 2007, Fairfax, VA, USA, November 2, 2007. pp. 22–30. ACM (2007), http://doi.acm.org/10.1145/1314436.1314440
4. Chlipala, A.: Certified Programming with Dependent Types. MIT Press (2011), http://adam.chlipala.net/cpdt/, http://adam.chlipala.net/cpdt/
5. Collier, G., Piccariello, H., Robson, R.: A digital rights management ecosystem model for the education community. ECAR (forthcoming), May (2004)
6. Halpern, J.Y., Weissman, V.: Using First-Order Logic to Reason about Policies. ACM Transactions on Information and System Security 11(4) (2008), http://doi.acm.org/10.1145/1380564.1380569
7. Holzer, M., Katzenbeisser, S., Schallhart, C.: Towards formal semantics for ODRL. In: Iannella, R., Guth, S. (eds.) Proceedings of the First International Workshop on

the Open Digital Rights Language (ODRL), Vienna, Austria, April 22-23, 2004. pp. 137–148 (2004), http://odrl.net/workshop2004/paper/odrl-katzenbeisser-paper.pdf
8. Iannella., R.: Open digital rights language (ODRL) version 1.1 (2002), [Online; accessed 09-February-2015]
9. Kasten, A., Grimm, R.: Making the semantics of ODRL and URM explicit using web ontologies. In: The 8th International Workshop for Technical, Economic and Legal Aspects of Business Models for Virtual Goods. pp. 77–91. Namur, Belgium (2010)
10. Maroñas, X., Rodríguez, E., Delgado, J.: An architecture for the interoperability between rights expression languages based on XACML. In: Proceedings of the 7th International Workshop for technical, economic and legal aspects of business models for virtual goods, Virtual Goods (2009)
11. Masi, M., Pugliese, R., Tiezzi, F.: Formalisation and implementation of the XACML access control mechanism. In: Barthe, G., Livshits, B., Scandariato, R. (eds.) Engineering Secure Software and Systems - 4th International Symposium, ESSoS 2012, Eindhoven, The Netherlands, February, 16-17, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7159, pp. 60–74. Springer (2012), http://dx.doi.org/10.1007/978-3-642-28166-2_7
12. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)
13. Prados, J., Rodriguez, E., Delgado, J.: Interoperability between different rights expression languages and protection mechanisms. In: Automated Production of Cross Media Content for Multi-Channel Distribution, 2005. AXMEDIS 2005. First International Conference on. pp. 8–pp. IEEE (2005)
14. Pucella, R., Weissman, V.: A formal foundation for ODRL. CoRR abs/cs/0601085 (2006), http://arxiv.org/abs/cs/0601085
15. Smalley, S., Vance, C., Labs, N., Salamon, W.: Implementing SELinux as a Linux security module. In: USENIX Technical Conference (2002), [Online; accessed 16-February-2015]
16. St-Martin, M., Felty, A.P.: A verified algorithm for detecting conflicts in XACML access control rules. In: Avigad, J., Chlipala, A. (eds.) Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016. pp. 166–175. ACM (2016), http://doi.acm.org/10.1145/2854065.2854079
17. Team, T.C.D.: The Coq Proof Assistant Reference Manual. LogiCal Project (2004), http://coq.inria.fr, version 8.0
18. Tschantz, M.C., Krishnamurthi, S.: Towards reasonability properties for access-control policy languages. In: Ferraiolo, D.F., Ray, I. (eds.) SACMAT 2006,11th ACM Symposium on Access Control Models and Technologies, Lake Tahoe, California, USA, June 7-9, 2006, Proceedings. pp. 160–169. ACM (2006), http://doi.acm.org/10.1145/1133058.1133081
19. Wang, X., Lao, G., DeMartini, T., Reddy, H., Nguyen, M., Valenzuela, E.: XrML - eXtensible rights markup language. In: Kudo, M. (ed.) Proceedings of the 2002 ACM Workshop on XML Security, Fairfax, VA, USA, November 22, 2002. pp. 71–79. ACM (2002), http://doi.acm.org/10.1145/764792.764803